Evolvable Integration of Activities with Statecharts

Gurcan Gulesir
University of Twente
gulesirg@cs.utwente.nl

Lodewijk Bergmans
University of Twente
bergmans@cs.utwente.nl

Mehmet Aksit
University of Twente
aksit@cs.utwente.nl

ABSTRACT

The dynamic behavior of a system can be specified in statecharts, and the activities of the system can be implemented in terms of functions in the C programming language. Later, the statecharts and the activities can be integrated to realize the system that fulfils a given set of requirements.

After the integration, the statecharts, the activities, and the requirements are subject to change due to emerging necessities such as bug fixes. Any change to any of these artifacts has a cost in terms of effort, and risk of errors.

In this paper, we provide a rigorous analysis of a relevant subset of possible changes to activities, and their associated costs. In addition, we present the overview of our solution to reduce these costs.

1. INTRODUCTION

Statecharts [18] can be used to specify the dynamic behavior of a system. Alternative methods such as object-oriented [24] and structured [8] analysis and design can be used to decompose the **activities**[18] of the system, and alternative programming languages such as Java [9] and C [20] can be used to implement the activities. The right choice depends on the characteristics of a given problem domain, because each alternative has certain strengths and weaknesses (chapter 1 in [8]). In this paper, we reason about those systems whose dynamic behavior is specified in statecharts, and whose activities are decomposed into functions that are implemented in the C programming language [20].

Once statecharts are specified, and the activities are implemented, then the statecharts and the activities should be integrated[1] to realize a system that fulfils a given set of requirements. After the integration, the set of requirements, the statecharts, and the activities are still subject to change, due to various emerging necessities [2] such as bug fixes. Any change to any of these artifacts involves a cost in terms of effort, and a risk of errors.

[1]This is explained in section 2.2 in detail.

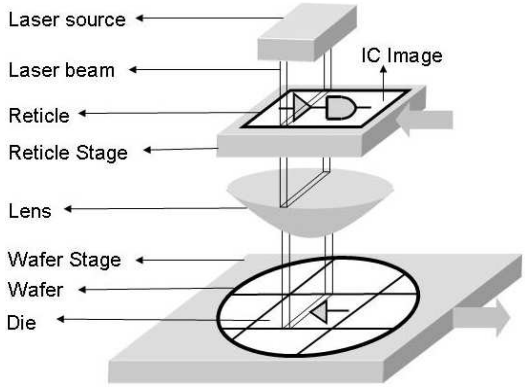

Figure 1: A snapshot of the wafer scanner during scanning.

In section 2, we explain a simplified version of a large-scale industrial application which we use to demonstrate our ideas; In section 4, we present the effort-consuming and error-prone nature of a relevant subset of possible changes to activities; And in section 5, we present a solution sketch for reducing the effort and the errors.

2. AN INDUSTRIAL APPLICATION

A **silicon wafer** is a circular slice of silicon used for producing integrated circuits (ICs). A **wafer scanner** is a semiconductor manufacturing equipment that exposes IC images onto silicon wafers. An ASML [5] wafer scanner is a large-scale embedded system that has 400 sensors, 300 actuators, 50 processors, and software containing 15 million lines of source code written in C programming language [20].

In this section, we present a simplified version of the wafer scanner and its software. We explain the activities the wafer scanner can perform, and describe how these activities are controlled by a statechart representing the wafer scanner's dynamic behavior.

2.1 Simplified Wafer Scanner

The wafer scanner (figure 1) uses a laser beam to **scan** an IC image, and uses a lens to **expose** the image on a rectangular region on the wafer. Such a region is called **die**.

2.1.1 Processing Activity

The material containing an IC image is called **reticle**. Before scanning, a reticle must be loaded onto a platform called **reticle stage**, and a wafer must be loaded onto a platform called **wafer stage**. Figure 1 shows a snapshot of the wafer scanner during **scanning** while the lens and the laser source are fixed, the laser source is emitting a laser beam, and the wafer stage and the reticle stage are moving in opposite directions. Consequently, the IC image is being exposed on a die. When the IC image is completely exposed, the laser source will be turned off, and the wafer stage will be moved to align the next die with the lens. This activity is called **advancing**. The **processing** activity is advancing to a die, then scanning it, and repeating this for each die on the wafer. This activity can be implemented as the source code in listing 1.

```
1     int i;
2     for(i = 0; i < numDies; i++)
3     {
4         advance(i);
5         scan();
6     }
7     return;
8 }
```

Listing 1: An implementation of the processing activity. The definitions of the global variable `numDies`, and the functions `advance` and `scan` are ommited, because they are irrelevant here.

2.1.2 Preprocessing Activity

To produce faultless ICs, the wafer scanner's actuators need to operate at a level of precision that is measured in terms of nanometers. To attain this precision level, two issues must be resolved: First, the reticle must be clean. Second, the wafer scanner must know the shape imperfections of the wafer, so that it can compensate for them during processing. Therefore, before processing, the wafer scanner must carry out the **preprocessing** activity that is **cleaning** the reticle if it is dirty, *and then* **measuring** the shape imperfections of the wafer. This activity can be implemented as the source code in listing 2.

```
9     if(!reticleIsClean)
10    {
11        cleanReticle();
12    }
13    measureWafer();
14    return;
15 }
```

Listing 2: An implementation of the preprocessing activity. The irrelevant details such as the definition of the global variable `reticleIsClean` are omitted.

2.1.3 Requirements

The requirements of the wafer scanner are

R1 The wafer scanner must start upon an external signal.
R2 The wafer scanner must process the wafer.
R3 After processing, all ICs on the wafer must be faultless.
R4 The wafer scanner must stop after the wafer is processed.

2.1.4 Dynamic Behavior

Considering the requirements, the dynamic behavior of the wafer scanner can be specified as the statechart in figure 2: If the scanner is in state READY when event start occurs (e.g. an operator presses the start button), then the scanner enters state PREPROCESSING where it starts the preprocessing activity (i.e. calls function `preprocess`). If the scanner is in state PREPROCESSING when event preprocessed occurs, then the scanner enters state PROCESSING where it starts the processing activity (i.e. calls function `process`). If the scanner is in state PROCESSING when event processed occurs, then the scanner enters the final state.

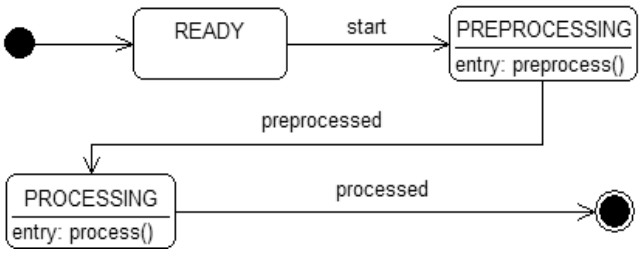

Figure 2: The wafer scanner's dynamic behavior.

Based on certain formal semantics of statecharts (e.g. [19]), transforming the statecharts to executable models is possible. This transformation is not a point of focus here. Therefore, we assume that the statechart (figure 2) has an underlying executable model, which operates as we have explained above.

Note that the statechart fulfils R1. However, R2, R3 and R4 are not fulfilled yet, because the activities and the statechart are not yet fully integrated. This is achieved in section 2.2.

2.2 Integrating Activities with Statecharts

Integration of activities with statecharts is two fold: a) Linking activities to states [18]. This is already done for our application (see figure 2). b) Reporting the events that occur as a result of execution of some functions that are the implementations of some activities or actions. This requires identifying all the points in the functions' source code where events occur during execution.

> ***Definition:*** An event e is **mapped** to a point p in source code, if and only if e occurs when an execution reaches p.

Considering the given definitions of the preprocessing and processing activities (see sections 2.1.2 and 2.1.1), the mappings of events preprocessed and processed can manually be identified: Event preprocessed is mapped to the point located after ';' in line 7, in listing 2. Event processed is mapped to the point located after '}' in line 8, in listing 1. After identification of the source code points, a function must be called (or some variables must be written) at each identified point to inform statecharts about the occurrence of the event, and to pass the relevant context information such as the values of the variables that are necessary for evaluating certain guards. We term such functions **event report**s. The listings 3 and 4 show the implementations of the preprocessing and processing activities after adding calls to event reports `reportPreprocessed` and `reportProcessed` at the identified points. The implementation details of the event reports are irrelevant here.

```
16    if(!reticleIsClean)
```

```
17      {
18          cleanReticle();
19      }
20      measureWafer();
21      reportPreprocessed();
22      return;
23  }
```

Listing 3: The preprocessing activity after its integration with the statechart (figure 2).

```
24      int i;
25      for(i = 0; i < numDies; i++)
26      {
27          advance(i);
28          scan();
29      }
30      reportProcessed();
31      return;
32  }
```

Listing 4: The processing activity after its integration with the statechart (figure 2).

Note that each requirement (section 2.1.3) is fulfilled by the wafer scanner that is the resulting system after the integration explained in this section.

3. DEFINITIONS

In this section, we define some terms that we use in the upcoming sections.

The **event reporting functionality** of a system is the functionality implemented by the calls to the event reports.

Let sc be the source code consisting of the implementations of all activities of a system. The event reporting functionality erf of the system is **sound** if and only if any call to any event report is located at a point in sc to which the related event is mapped. erf is **complete** if and only if at any point in sc to which an event is mapped, a call to the related event report exists[2].

According to Harel et. al. [19], "The behavior of a system is a set of possible **runs**, each representing the responses of the system to a sequence of external stimuli generated by its environment."

3.1 Call Flow Graph

Let f be a function definition in the C programming language. $callsFrom(f)$ denotes the set of function calls whose **static scope** [23] is the definition of f (i.e. the calls to functions from the body of f). For example, let cr and mw respectively denote the calls to functions `cleanReticle` and `measureWafer` in listing 2. $callsFrom($`preprocess`$) = \{cr, mw\}$. Let G be the **control flow graph** [14] derived from the definition of f. $firstCall(c, f)$ denotes that c is a first function call according to G. For example, $firstCall(cr,$ `preprocess`$)$ is true. $nextCall(c_1, c_2, f)$ denotes that c_1 is a next function call after function call c_2, according to G. For example, $nextCall(mw, cr,$`preprocess`$)$ is true. $lastCall(c, f)$ denote that c is a last function call according to G. For example, $lastCall(mw,$`preprocess`$)$ is true. $noCall(f)$ denotes that there is at least one **path** [14] p in G such that $\nexists c(c \in callsFrom(f) \wedge c \in p)$. $name(c)$ denotes the **identifier** [23] of the C function one of whose call is c.

CFG_f denotes the **call flow graph** of f, which is a tuple $\langle V = \{\nu_0\} \cup V_I \cup \{\nu_F\}, E = E_0 \cup E_I \cup E_F \rangle$, where

- ν_0 is the **initial node**.
- V_I is a finite set of **internal nodes** such that $\nu_0 \notin V_I$, and a bijection $nodeOfCall : callsFrom(f) \rightarrow V_I$ exists. The label of any $\nu \in V_I$ is $name(nodeOfCall^{-1}(\nu))$.
- ν_F is the **final node** such that $\nu_F \neq \nu_0$ and $\nu_F \notin V_I$.
- $E_0 \subseteq \{\nu_0\} \times (V_I \cup \{\nu_F\})$ is the set of **initial edges** such that $\forall c(firstCall(c, f) \Rightarrow (\nu_0, nodeOfCall(c)) \in E_0)$, and $noCall(f) \Rightarrow (\nu_0, \nu_f) \in E_0$.
- $E_I \subseteq V_I \times V_I$ is a set of **internal edges** such that $\forall \nu_1, \nu_2, c_1, c_2(nodeOfCall(c_1) = \nu_1 \wedge nodeOfCall(c_2) = \nu_2 \wedge nextCall(c_1, c_2, f) \Rightarrow (\nu_2, \nu_1) \in E_I)$.
- $E_F \subseteq (V_I \cup \{\nu_0\}) \times \{\nu_F\}$ is the set of **final edges** such that $\forall c(lastCall(c) \Rightarrow (nodeOfCall(c), \nu_F) \in E_F)$, and $noCall(f) \Rightarrow (\nu_0, \nu_f) \in E_F$.

The call flow graph of an activity's implementation f can be constructed by traversing the **abstract syntax tree** (AST) [6] rooted at the signature of f.

A **call flow path** p of a function f is a path in the call flow graph of f such that ν_0 and ν_f are respectively the first and the last nodes lying on p. For example, function `preprocess` (listing 2) has two call-flow paths: ν_0, `measureWafer`, ν_F; and ν_0, `cleanReticle`, `measureWafer`, ν_F.

4. DIFFICULTY OF EVOLUTION

Software evolves due to new requirements, bug fixes, qualitative improvements, etc. [2]. Making mistakes during evolution results in systems that do not fulfill some of their requirements. In this section, we present some changes to the implementation of the wafer scanner's activities, which result in defects in the integration of the activities with the statechart. These defects prevent the wafer scanner from fulfilling some of its requirements. We do not discuss the evolution of statecharts and its associated difficulty here; we leave it as future work.

4.1 Difficulty of Removing a Function Call

If we remove the call to function `measureWafer` in line 7 in listing 3, then the call to event report `reportPreprocessed` will be located at a point to which event `preprocessed` is no longer mapped, because the wafer is not measured before that point. Hence, the event reporting functionality will no longer be sound, and requirement R3 will not be fulfilled: the processing activity starts before the preprocessing activity is completed, resulting in defective ICs. To avoid this, we must remove the call to event report `reportPreprocessed`. In this case however, requirements R2 and R4 will not be fulfilled, because the statechart will never be stimulated with an occurrence of event `preprocessed`. Thus, we can conclude that removing the call to `measureWafer` is forbidden according to the requirements and the wafer scanner's behavioral design (i.e. the structure of the statechart in figure 2, the linking of the activities to states, and the labelling of the transitions).

4.1.1 Constraints on Activities

[2]If multiple events are mapped to a point, then an ordering (e.g. [22]) among the related event reports is necessary.

Based on the wafer scanner's behavioral design, requirements R2 and R4 constrain that event preprocessed must occur any time function `preprocess` is executed. Otherwise, the wafer scanner cannot perform the transitions from state PREPROCESSING to state PROCESSING, and from state PROCESSING to the final state, in at least one run. Consequently, requirements R2 and R4 cannot be fulfilled in those runs.

Based on the given definition of the preprocessing activity (see section 2.1.2), the constraint on function `preprocess` can be decomposed into finer-grained constraints:

C1: At least one `measureWafer` node must lie on any call flow path of `preprocess`.
C2: Any `measureWafer` node must not lie before any `cleanReticle` node on any call flow path of `preprocess`.

Similarly, requirement R4 constrains that event processed must occur any time function `process` is executed. This constraint can also be decomposed into finer-grained constraints:

C3: At least one `advance` node must lie on any call flow path of `process`.
C4: At least one `scan` node must lie on any call flow path of `process`.
C5: On any call flow path p of `process` such that at least one `advance` node and at least one `scan` node lie on p, the first `advance` node must lie before the first `scan` node.
C6: On any call flow path p of `process` such that at least one `advance` node and at least one `scan` node lie on p, the last `advance` node must lie before the last `scan` node.
C7: On any call flow path of `process`, exactly one `scan` node must lie between any two consecutive `advance` nodes.
C8: On any call flow path of `process`, exactly one `advance` node must lie between any two consecutive `scan` nodes.

4.2 Difficulty of Adding a Function Call

Adding a function call may result in a new mapping of an existing type of event (e.g. preprocessed) to a point in the source code, in which case the event reporting functionality becomes incomplete. To restore completeness, adding a call to the related event report at the source code point is necessary. Otherwise, the system cannot sense some occurrences of the event, and react to them. Consequently, some requirements may not be fulfilled.

4.3 Difficulty of Adding a Control Statement

Consider a new requirement indicating the wafer must be measured only if the reticle is clean. To fulfill the requirement, we can 'wrap' the call to function `measureWafer` (line 7, listing 3) with an `if` block, as shown in listing 5.

```
1  void preprocess()
2  {
3      if(!reticleIsClean)
4      {
5          cleanReticle();
6      }
7      if(reticleIsClean)
8      {
9          measureWafer();
10     }
11     reportPreprocessed();
12     return;
13 }
```

Listing 5: The preprocessing activity after adding a new control statement.

In this case, the call to event report `reportPreprocessed` (line 11) is located at a point to which event preprocessed is not mapped. Thus, the event reporting functionality is unsound. In addition, event preprocessed is mapped to the point located after `;` in line 9 where a call to event report `reportPreprocessed` does not exist. Hence, the event reporting functionality is incomplete. To restore soundness and completeness, we must move the call to event report `reportPreprocessed` from line 11 to the point located after `;` in line 9. Otherwise, requirement R3 may not be fulfilled.

4.4 Difficulty of Restructuring

Consider an extract-function restructuring [17] that involves moving lines 3-7 in listing 3 to a new function `newPreprocess`, as in listing 6.

```
1  void preprocess()
2  {
3      newPreprocess();
4      reportPreprocessed();
5      return;
6  }
7  void newPreprocess()
8  {
9      if(!reticleIsClean)
10     {
11         cleanReticle();
12     }
13     measureWafer();
14     return;
15 }
```

Listing 6: An extract-function restructuring.

In this case, the event reporting functionality is both unsound and incomplete, similar to the case in section 4.3. Nevertheless, all the requirements are still fulfilled, in contrast to the cases in sections 4.1, 4.2, and 4.3. This is certainly what is expected from a restructuring by definition. However, if the system evolves further, and function `newPreprocess` is called from an additional place different than line 4, then the system will not be able to sense some occurrences of event preprocessed, and react to them. If the call to event report `reportPreprocess` in line 4 is moved to the point located after `;` in line 13, then the soundness and completeness will be restored.

4.5 Summary of the Difficulty of Evolution

Each time activities evolve, checking whether the constraints on the activities are satisfied, in addition to maintaining a sound and complete event reporting functionality is necessary. If these tasks are manually carried out, they are effort-consuming and error-prone. This is the problem we address in this paper.

5. SOLUTION OVERVIEW

We propose a 3-stage solution (figure 3) to automatically a) check whether the activities' implementations satisfy the related constraints, b) add calls to the event reports in the source code, such that the event reporting functionality is sound and complete.

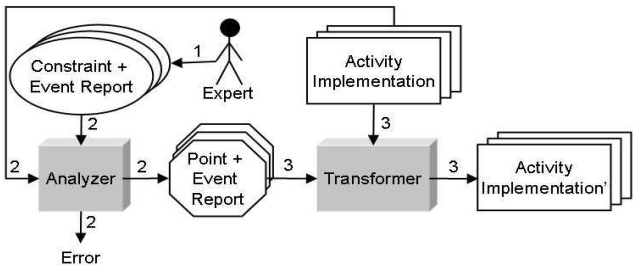

Figure 3: A 3-stage solution to automatically check whether the constraints are satisfied, and to automatically implement the event reporting functionality.

At the first stage an expert specifies the constraints on the activities, and then associates the relevant event reports with the relevant elements in the constraints.

At the second stage, an analysis tool is provided with the output of the expert from stage 1, and the implementations of the activities (e.g. listings 1 and 2) of the system. As a result of the analysis, if the constraints are satisfied, the analyzer outputs the points in the source code to which the events are mapped, together with the event reports that must be called at each output point. If the constraints are not satisfied (e.g. the situation explained in section 4.1), the analyzer gives an error.

At the third stage, a source code transformation tool is provided with the activities' source code and the output of the analysis tool from stage 2. The transformer adds calls to the event reports at the identified points in the activities' source code. This transformation results in a sound and complete event reporting functionality.

If the activities evolve, stages 2 and 3 can automatically be repeated a) to check whether the activities' implementations satisfy the related constraints, and b) to re-add calls to the event reports so that the event reporting functionality remains sound and complete. Hence, effort can be saved and errors can be reduced.

6. RELATED WORK

Because we have not provided any details about our solution, a rigorous comparison with the related work is not feasible here. Nevertheless, we can provide the following information based on what we have already explained.

The work presented in this paper relates to various facets of aspect-oriented software development. The source code points mentioned in section 2 are equivalent to the concept of **joinpoint** [3], and the event reports mentioned in section 5 are equivalent to the concept of **advice** [4]. Although the constraints mentioned in section 5 are similar to the concept of **pointcut designator** [4], they have a difference: They not only describe a set of joinpoints, but also constrain the operational semantics of the functions to which they are bound.

Intuitively, one can think of a constraint as a combination of a) A trace-based pointcut designator [10, 13, 11, 12, 7] that essentially does regular language-based pattern matching to identify the joinpoints of interest. b) A **declare error** [1] like pointcut designator that checks certain properties of the operational semantics of the functions. We statically analyze the possible traces of functions to identify joinpoints, whereas approaches like [12, 7] monitor the observable trace of a programs during execution. Due to its property checking feature, our work relates to [16, 21], too.

7. CONCLUSIONS

We presented a practical real-life problem based on a large-scale industrial application. In addition, we provided the overview of our solution that has many similarities with some of the recognized aspect-oriented approaches. Hence, we demonstrated a case in which aspect-orientation is beneficial beyond the classical cases such as logging and tracing.

8. ACKNOWLEDGMENTS

This work has been carried out as a part of the Ideals project under the management of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

9. REFERENCES

[1] AspectJ TM 5 Development Kit Developer's Notebook. http://www.eclipse.org/aspectj/doc/released/adk15notebook/.

[2] *IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society.

[3] Introduction. In Filman et al. [15], page 4.

[4] Introduction. In Filman et al. [15], page 5.

[5] ASML. http://www.asml.com.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364. ACM Press, 2005.

[8] G. Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[9] N. B. Dale, C. Weems, and M. Headington. *Introduction to Java and Software Design*. Jones and Bartlett Publishers, Inc., , USA, 2000.

[10] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 173–188, London, UK, 2002. Springer-Verlag.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[12] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In Filman et al. [15], pages 201–217.

[13] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 170–186, London, UK, 2001. Springer-Verlag.

[14] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[15] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[16] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[17] W. G. Griswold and D. Notkin. Automated Assistance for Program Restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[19] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[20] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[21] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[22] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *Proceedings of International Conference NetObjectDays, NODe2005*, Sep 2005.

[23] R. W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, 4 edition, 1999.

[24] E. Yourdon. *Modern structured analysis*. Yourdon Press, Upper Saddle River, NJ, USA, 1989.